

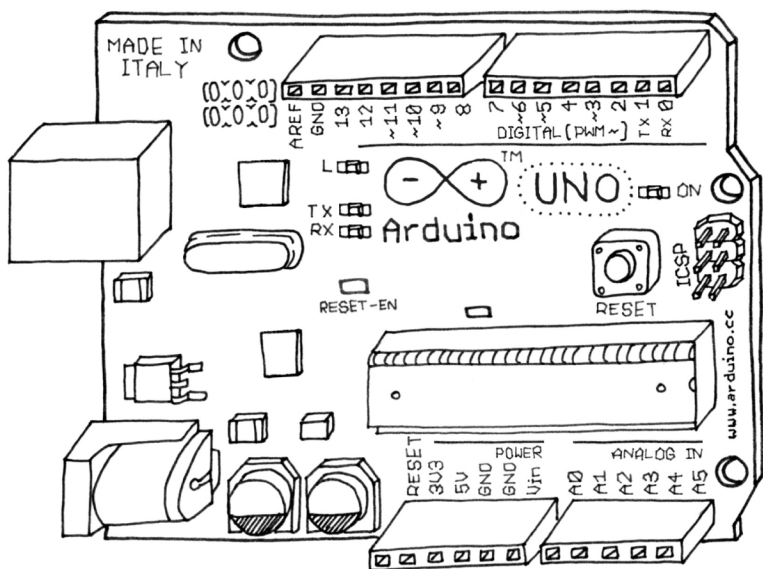
Make: PROJECTS

THE OPEN
SOURCE
ELECTRONICS
PROTOTYPING
PLATFORM

Getting Started with Arduino

2nd Edition

Massimo Banzi co-founder of Arduino



O'REILLY

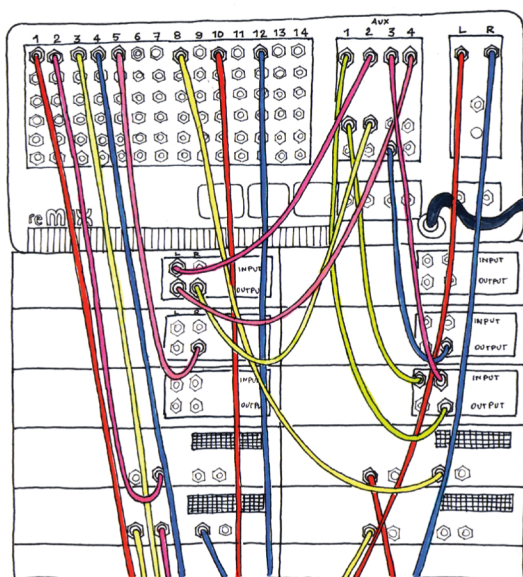
Make:
makezine.com

www.it-ebooks.info

Arduino is the open source electronics prototyping platform that's taken the design and hobbyist world by storm. This thorough introduction, updated for Arduino 1.0, gives you lots of ideas for projects and helps you work with them right away. From getting organized to putting the final touches on your prototype, all the information you need is here!

Inside, you'll learn about:

- » Interaction design and physical computing
- » The Arduino hardware and software development environment
- » Basics of electricity and electronics
- » Prototyping on a solderless breadboard
- » Drawing a schematic diagram



Getting started with Arduino is a snap. To use the introductory examples in this guide, all you need is an Arduino Uno or earlier model, along with a USB A-B cable and an LED. The easy-to-use Arduino development environment is free to download.

Join hundreds of thousands of hobbyists who have discovered this incredible (and educational) platform. Written by the co-founder of the Arduino project, *Getting Started with Arduino* gets you in on all the fun!

Illustrations by Elisa Canducci with Shawn Wallace

Make:
makezine.com

O'REILLY

US \$14.99

CAN \$15.99

ISBN: 978-1-449-30987-9



9

Appendix C/Arduino Quick Reference

Here is a quick explanation of all the standard instructions supported by the Arduino language.

For a more detailed reference, see: arduino.cc/en/Reference/HomePage

STRUCTURE

An Arduino sketch runs in two parts:

`void setup()`

This is where you place the initialisation code—the instructions that set up the board before the main loop of the sketch starts.

`void loop()`

This contains the main code of your sketch. It contains a set of instructions that get repeated over and over until the board is switched off.

SPECIAL SYMBOLS

Arduino includes a number of symbols to delineate lines of code, comments, and blocks of code.

; (semicolon)

Every instruction (line of code) is terminated by a semicolon. This syntax lets you format the code freely. You could even put two instructions on the same line, as long as you separate them with a semicolon. (However, this would make the code harder to read.)

Example:

```
delay(100);
```

{ } (curly braces)

This is used to mark blocks of code. For example, when you write code for the `loop()` function, you have to use curly braces before and after the code.

Example:

```
void loop() {  
    Serial.println("ciao");  
}
```

comments

These are portions of text ignored by the Arduino processor, but are extremely useful to remind yourself (or others) of what a piece of code does.

There are two styles of comments in Arduino:

```
// single-line: this text is ignored until the end of the line
/* multiple-line:
   you can write
   a whole poem in here
*/
```

CONSTANTS

Arduino includes a set of predefined keywords with special values.

HIGH and **LOW** are used, for example, when you want to turn on or off an Arduino pin. **INPUT** and **OUTPUT** are used to set a specific pin to be either an input or an output.

true and **false** indicate exactly what their names suggest: the truth or falsehood of a condition or expression.

VARIABLES

Variables are named areas of the Arduino's memory where you can store data that you can use and manipulate in your sketch. As the name suggests, they can be changed as many times as you like.

Because Arduino is a very simple processor, when you declare a variable you have to specify its type. This means telling the processor the size of the value you want to store.

Here are the *datatypes* that are available:

boolean

Can have one of two values: true or false.

char

Holds a single character, such as A. Like any computer, Arduino stores it as a number, even though you see text. When chars are used to store numbers, they can hold values from -128 to 127.

NOTE: There are two major sets of characters available on computer systems: ASCII and UNICODE. ASCII is a set of 127 characters that was used for, among other things, transmitting text between serial terminals and time-shared computer systems such as mainframes and minicomputers. UNICODE is a much larger set of values used by modern computer operating systems to represent characters in a wide range of languages. ASCII is still useful for exchanging short bits of information in languages such as Italian or English that use Latin characters, Arabic numerals, and common typewriter symbols for punctuation and the like.

byte

Holds a number between 0 and 255. As with chars, bytes use only one byte of memory.

int

Uses 2 bytes of memory to represent a number between -32,768 and 32,767; it's the most common data type used in Arduino.

unsigned int

Like int, uses 2 bytes but the **unsigned** prefix means that it can't store negative numbers, so its range goes from 0 to 65,535.

long

This is twice the size of an **int** and holds numbers from -2,147,483,648 to 2,147,483,647.

unsigned long

Unsigned version of **long**; it goes from 0 to 4,294,967,295.

float

This quite big and can hold floating-point values, a fancy way of saying that you can use it to store numbers with a decimal point in it. It will eat up 4 bytes of your precious RAM and the functions that can handle them use up a lot of code memory as well. So use **floats** sparingly.

double

Double-precision floating-point number, with a maximum value of $1.7976931348623157 \times 10^{308}$. Wow, that's huge!

string

A set of ASCII characters that are used to store textual information (you might use a string to send a message via a serial port, or to display on

an LCD display). For storage, they use one byte for each character in the string, plus a null character to tell Arduino that it's the end of the string. The following are equivalent:

```
char string1[] = "Arduino"; // 7 chars + 1 null char
char string2[8] = "Arduino"; // Same as above
```

array

A list of variables that can be accessed via an index. They are used to build tables of values that can easily be accessed. For example, if you want to store different levels of brightness to be used when fading an LED, you could create six variables called light01, light02, and so on. Better yet, you could use a simple array like:

```
int light[6] = {0, 20, 50, 75, 100};
```

The word "array" is not actually used in the variable declaration: the symbols [] and {} do the job.

CONTROL STRUCTURES

Arduino includes keywords for controlling the logical flow of your sketch.

if . . . else

This structure makes decisions in your program. *if* must be followed by a question specified as an expression contained in parentheses. If the expression is true, whatever follows will be executed. If it's false, the block of code following *else* will be executed. It's possible to use just *if* without providing an *else* clause.

Example:

```
if (val == 1) {
    digitalWrite(LED,HIGH);
}
```

for

Lets you repeat a block of code a specified number of times.

Example:

```
for (int i = 0; i < 10; i++) {
    Serial.print("ciao");
}
```

switch case

The *if* statement is like a fork in the road for your program. *switch case* is like a massive roundabout. It lets your program take a variety of directions

depending on the value of a variable. It's quite useful to keep your code tidy as it replaces long lists of *if* statements.

Example:

```
switch (sensorValue) {
    case 23:
        digitalWrite(13,HIGH);
        break;
    case 46:
        digitalWrite(12,HIGH);
        break;
    default: // if nothing matches this is executed
        digitalWrite(12,LOW);
        digitalWrite(13,LOW);
}
```

while

Similar to *if*, this executes a block of code while a certain condition is true.

Example:

```
// blink LED while sensor is below 512
sensorValue = analogRead(1);
while (sensorValue < 512) {
    digitalWrite(13,HIGH);
    delay(100);
    digitalWrite(13,HIGH);
    delay(100);
    sensorValue = analogRead(1);
}
```

do . . . while

Just like *while*, except that the code is run just before the the condition is evaluated. This structure is used when you want the code inside your block to run at least once before you check the condition.

Example:

```
do {
    digitalWrite(13,HIGH);
    delay(100);
    digitalWrite(13,HIGH);
    delay(100);
    sensorValue = analogRead(1);
} while (sensorValue < 512);
```

break

This term lets you leave a loop and continue the execution of the code that appears after the loop. It's also used to separate the different sections of a *switch* case statement.

Example:

```
// blink LED while sensor is below 512
do {
    // Leaves the loop if a button is pressed
    if (digitalRead(7) == HIGH)
        break;
    digitalWrite(13,HIGH);
    delay(100);
    digitalWrite(13,LOW);
    delay(100);
    sensorValue = analogRead(1);
} while (sensorValue < 512);
```

continue

When used inside a loop, *continue* lets you skip the rest of the code inside it and force the condition to be tested again.

Example:

```
for (light = 0; light < 255; light++)
{
    // skip intensities between 140 and 200
    if ((x > 140) && (x < 200))
        continue;
    analogWrite(PWMPin, light);
    delay(10);
}
```

return

Stops running a function and returns from it. You can also use this to return a value from inside a function.

For example, if you have a function called *computeTemperature()* and you want to return the result to the part of your code that invoked the function you would write something like:

```
int computeTemperature() {
    int temperature = 0;
    temperature = (analogRead(0) + 45) / 100;
    return temperature;
}
```

ARITHMETIC AND FORMULAS

You can use Arduino to make complex calculations using a special syntax. + and – work like you’ve learned in school, and multiplication is represented with an * and division with a /.

There is an additional operator called “modulo” (%), which returns the remainder of an integer division. You can use as many levels of parentheses as necessary to group expressions. Contrary to what you might have learned in school, square brackets and curly brackets are reserved for other purposes (array indexes and blocks, respectively).

Examples:

```
a = 2 + 2;
light = ((12 * sensorValue) - 5) / 2;
remainder = 3 % 2; // returns 1
```

COMPARISON OPERATORS

When you specify conditions or tests for *if*, *while*, and *for* statements, these are the operators you can use:

- == equal to
- != not equal to
- < less than
- > greater than
- <= less than or equal to
- >= greater than or equal to

BOOLEAN OPERATORS

These are used when you want to combine multiple conditions. For example, if you want to check whether the value coming from a sensor is between 5 and 10, you would write:

```
if ((sensor == 5) && (sensor <=10))
```

There are three operators: and, represented with **&&**; or, represented with **||**; and finally not, represented with **!**.

COMPOUND OPERATORS

These are special operators used to make code more concise for some very common operations like incrementing a value.

For example, to increment *value* by 1 you would write:

```
value = value +1;
```

but using a compound operator, this becomes:

```
value++;
```

increment and decrement (-- and ++)

These increment or decrement a value by 1. Be careful: if you write *i++* this increments *i* by 1 and evaluates to the equivalent of *i+1*; *++i* evaluates to the value of *i* and **then** increments *i*. The same applies to *--*.

+=, -=, *= and /=

These make it shorter to write certain expressions. The following two expressions are equivalent:

```
a = a + 5;  
a += 5;
```

INPUT AND OUTPUT FUNCTIONS

Arduino includes functions for handling input and output. You've already seen some of these in the example programs throughout the book.

pinMode(pin, mode)

Reconfigures a digital pin to behave either as an input or an output.

Example:

```
pinMode(7,INPUT); // turns pin 7 into an input
```

digitalWrite(pin, value)

Turns a digital pin either on or off. Pins must be explicitly made into an output using *pinMode* before *digitalWrite* will have any effect.

Example:

```
digitalWrite(8,HIGH); // turns on digital pin 8
```

int digitalRead(pin)

Reads the state of an input pin, returns HIGH if the pin senses some voltage or LOW if there is no voltage applied.

Example:

```
val = digitalRead(7); // reads pin 7 into val
```

int analogRead(pin)

Reads the voltage applied to an analog input pin and returns a number between 0 and 1023 that represents the voltages between 0 and 5 V.

Example:

```
val = analogRead(0); // reads analog input 0 into val
```

analogWrite(pin, value)

Changes the PWM rate on one of the pins marked PWM. *pin* may be 11,10, 9, 6, 5, 3. *value* may be a number between 0 and 255 that represents the scale between 0 and 5 V output voltage.

Example:

```
analogWrite(9,128); // Dim an LED on pin 9 to 50%
```

shiftOut(dataPin, clockPin, bitOrder, value)

Sends data to a *shift register*, devices that are used to expand the number of digital outputs. This protocol uses one pin for data and one for clock. *bitOrder* indicates the ordering of bytes (least significant or most significant) and *value* is the actual byte to be sent out.

Example:

```
shiftOut(dataPin, clockPin, LSBFIRST, 255);
```

unsigned long pulseIn(pin, value)

Measures the duration of a pulse coming in on one of the digital inputs. This is useful, for example, to read some infrared sensors or accelerometers that output their value as pulses of changing duration.

Example:

```
time = pulseIn(7,HIGH); // measures the time the next
                        // pulse stays high
```

TIME FUNCTIONS

Arduino includes functions for measuring elapsed time and also for pausing the sketch.

unsigned long millis()

Returns the number of milliseconds that have passed since the sketch started.

Example:

```
duration = millis()-lastTime; // computes time elapsed since "lastTime"
```

delay(ms)

Pauses the program for the amount of milliseconds specified.

Example:

```
delay(500); // stops the program for half a second
```

delayMicroseconds(us)

Pauses the program for the given amount of microseconds.

Example:

```
delayMicroseconds(1000); // waits for 1 millisecond
```

MATH FUNCTIONS

Arduino includes many common mathematical and trigonometric functions:

min(x, y)

Returns the smaller of *x* and *y*.

Example:

```
val = min(10,20); // val is now 10
```

max(x, y)

Returns the larger of *x* and *y*.

Example:

```
val = max(10,20); // val is now 20
```

abs(x)

Returns the absolute value of *x*, which turns negative numbers into positive. If *x* is 5 it will return 5, but if *x* is -5, it will still return 5.

Example:

```
val = abs(-5); // val is now 5
```

constrain(x, a, b)

Returns the value of *x*, constrained between *a* and *b*. If *x* is less than *a*, it will just return *a* and if *x* is greater than *b*, it will just return *b*.

Example:

```
val = constrain(analogRead(0), 0, 255); // reject values bigger than 255
```

map(value, fromLow, fromHigh, toLow, toHigh)

Maps a value in the range *fromLow* and *maxLow* to the range *toLow* and *toHigh*. Very useful to process values from analogue sensors.

Example:

```
val = map(analogRead(0),0,1023,100, 200); // maps the value of
                                           // analog 0 to a value
                                           // between 100 and 200
```

double pow(base, exponent)

Returns the result of raising a number (*base*) to a value (*exponent*).

Example:

```
double x = pow(y, 32); // sets x to y raised to the 32nd power
```

double sqrt(x)

Returns the square root of a number.

Example:

```
double a = sqrt(1138); // approximately 33.73425674438
```

double sin(rad)

Returns the sine of an angle specified in radians.

Example:

```
double sine = sin(2); // approximately 0.90929737091
```

double cos(rad)

Returns the cosine of an angle specified in radians.

Example:

```
double cosine = cos(2); // approximately -0.41614685058
```

double tan(rad)

Returns the tangent of an angle specified in radians.

Example:

```
double tangent = tan(2); // approximately -2.18503975868
```

RANDOM NUMBER FUNCTIONS

If you need to generate random numbers, you can use Arduino's pseudo-random number generator.

randomSeed(seed)

Resets Arduino's pseudorandom number generator. Although the distribution of the numbers returned by *random()* is essentially random, the sequence is predictable. So, you should reset the generator to some random value. If you have an unconnected analog pin, it will pick up random noise from the surrounding environment (radio waves, cosmic rays, electromagnetic interference from cell phones and fluorescent lights, and so on).

Example:

```
randomSeed(analogRead(5)); // randomize using noise from pin 5
```

long random(max)

long random(min, max)

Returns a pseudorandom *long* integer value between *min* and *max* - 1.

If *min* is not specified, the lower bound is 0.

Example:

```
long randnum = random(0, 100); // a number between 0 and 99
long randnum = random(11);     // a number between 0 and 10
```

SERIAL COMMUNICATION

As you saw in Chapter 5, you can communicate with devices over the USB port using a serial communication protocol. Here are the serial functions.

Serial.begin(speed)

Prepares Arduino to begin sending and receiving serial data. You'll generally use 9600 bits per second (bps) with the Arduino IDE serial monitor, but other speeds are available, usually no more than 115,200 bps.

Example:

```
Serial.begin(9600);
```

Serial.print(data)

Serial.print(data, encoding)

Sends some data to the serial port. The encoding is optional; if not supplied, the data is treated as much like plain text as possible.

Examples:

```
Serial.print(75);           // Prints "75"
Serial.print(75, DEC);      // The same as above.
Serial.print(75, HEX);      // "4B" (75 in hexadecimal)
Serial.print(75, OCT);      // "113" (75 in octal)
Serial.print(75, BIN);      // "1001011" (75 in binary)
Serial.print(75, BYTE);     // "K" (the raw byte happens to
                             // be 75 in the ASCII set)
```

Serial.println(data)

Serial.println(data, encoding)

Same as *Serial.print()*, except that it adds a carriage return and linefeed (`\r\n`) as if you had typed the data and then pressed Return or Enter.

Examples:

```
Serial.println(75);         // Prints "75\r\n"
Serial.println(75, DEC);    // The same as above.
Serial.println(75, HEX);    // "4B\r\n"
Serial.println(75, OCT);    // "113\r\n"
Serial.println(75, BIN);    // "1001011\r\n"
Serial.println(75, BYTE);   // "K\r\n"
```

int Serial.available()

Returns how many unread bytes are available on the Serial port for reading via the *read()* function. After you have *read()* everything available, *Serial.available()* returns 0 until new data arrives on the serial port.

Example:

```
int count = Serial.available();
```

int Serial.read()

Fetches one byte of incoming serial data.

Example:

```
int data = Serial.read();
```

Serial.flush()

Because data may arrive through the serial port faster than your program can process it, Arduino keeps all the incoming data in a buffer. If you need to clear the buffer and let it fill up with fresh data, use the *flush()* function.

Example:

```
Serial.flush();
```